

# *EJB3*

An introduction and exploration of EJB3

Darin Manica

---

---

# *Why EJB3?*

- What does EJB3 need to provide?
- Why EJB3 instead of competing frameworks?



# ***EJB3 Features***

- All beans are homeless
  - No required subclassing or thrown Exceptions
  - JTA transactions
  - JAAS security
  - Dependency Injection
  - Interceptors
  - Callbacks and Callback Listeners
  - Java5 annotations with optional XML descriptors
  - Convention over configuration
  - Interop with EJB2.x
- 
-

# Overview

- Bean Types
    - Stateless and Stateful Session Beans
    - Message Driven Beans
    - Service POJOs (JBoss Extension)
    - Entity Beans
  - Dependency Injection & JNDI
  - Callbacks
  - Interceptors
  - Security
  - Deployment
- 
-

# *Let's Begin...*

- Bean Types
  - Stateless and Stateful Session Beans
  - Message Driven Beans
  - Service POJOs (JBoss Extension)
  - Entity Beans



# *Stateless Session Bean*

- Bean class
  - Annotate a POJO with @Stateless
- Local interface
  - Annotate an interface with @Local
- Remote interface
  - Annotate an interface with @Remote
- Optional
  - Annotate the Bean class with the remote and local interface



# ***SLSB Example***

```
@Stateless
@Local(HelloWorld.class)
public class HelloWorldServiceBean
    implements HelloWorld {
    public String sayHello(String name){
        return "Hello, " + name;
    }
}

public interface HelloWorld {
    String sayHello(String name);
}
```

---

---

# *Stateful Session Bean*

- Bean class
  - Annotate a POJO with `@Stateful`
- Instance variables are now allowed
- `@Remove`
  - Instead of calling `EJBObject.remove()`, simply annotate a `@Remove` method. The bean will be removed at the end of the call.



```
@Stateful public class HelloWorldServiceBean
    implements HelloWorld
{
    private String name;

    public String getName() {...}
    public void setName(String name) {...}

    public String sayHello() {
        return "Hello, " + name;
    }

    @Remove
    public void goodbye() {}
}

@Local public interface HelloWorld {
    public String sayHello();
}
```

---

---

# *Message Driven Bean*

- Very similar to before
  - Already homeless
  - Already implement `MessageListener`
- Implement `onMessage(Message)`
- Deployment concerns in annotations



# ***MDB Example***

```
@MessageDriven(activateConfig = {
    @ActivationConfigProperty(
        propertyName="destinationType",
        propertyValue="javax.jms.Queue"),
    @ActivationConfigProperty(
        propertyName="destination",
        propertyValue="queue/myque")})
public class MyMDB implements MessageListener {
    public void onMessage(Message message) {
        System.out.println("Message received");
    }
}
```

---

---

# *Service POJOs (JBoss only)*

- Mbean as a POJO
  - Annotate with @Service
  - Interfaces
    - @Management
    - @Local
    - @Remote
  - Responds to JBoss lifecycle events
    - Create
    - Start
    - Stop
    - Destroy
- 
-

# *Service POJO example*

```
@Service public class MyServiceMBean
    implements MyService
{
    public String saySomethingUseful()
    {
        System.out.println("Something useful");
    }
}
```

```
@Management public interface MyService {
    String saySomethingUseful();
}
```

---

---

# *Entity Beans Are...*

- Completely overhauled from EJB 2.x
  - POJOs allocated with *new*
  - Can be attached, detached and reattached to persistent storage
  - Optimistic locking
  - Can generate or update the schema in the database
    - Schema is defined via annotations
    - Can update or drop/add the schema
- 
-

# *Entity Beans Aren't...*

- No CMR
  - All relationships must be managed manually
- Entity Beans are not remotable



# *Creating Entity Beans*

- Annotate a POJO with @Entity
  - Define an id with @Id
    - Define generate. Usually GeneratorType.AUTO
  - All getter/setter pairs become persistent attributes
    - Any additional column annotations are placed on the getter
    - Getter/setters can use any access, including private
- 
-

# *Entity Bean Example*

```
@Entity public class Book
{
    private Integer id;
    private String name;
    private String isbn;

    @Id(generate = GenerationType.AUTO)
    public Integer getId() {...}
    public void setId(Integer id) {...}

    ...
}
```

---

---

# *Entity Schema Generation*

- All entities become tables
  - All getter become columns
  - Foreign keys created for relationships
  - Associative entities are created for @ManyToMany relationships
  - Annotations can override
    - Naming
    - Datatype (Calendar, enum, etc.)
    - Length
    - Nullability
    - Transience (@Transient)
- 
-

# *Entity Manager*

- Handles all interactions with the database
  - Find entities by primary key
  - Query over entities
- Provides CRUD functionality
- Injected with `@PersistenceContext`



# Optimistic Locking

- Annotate with @Version
- A failed optimistic lock rolls back the transaction

```
@Entity public class Flight implements Serializable
{
    ...
    @Version
    @Column(name="OPTLOCK")
    public Integer getVersion() {...}
}
```

# *Relationships: One to One*

- Annotate a getter with @OneToOne

```
@Entity public class Customer implements Serializable
{
    private Address address;

    @OneToOne
    public Address getAddress()
    { return this.address; }

    ...
}
```

# *Relationships: One to Many*

- Parent
    - Specify a collection (using generics) of children on the parent
    - Annotate the getter with @OneToMany
      - Bidirectional relationship must specify mappedBy which is the child's pointer to the parent.
  - Child
    - Specify a pointer back to the Parent
    - Annotate the getter
      - @ManyToOne
- 
-

# Relationships: One to Many

```
@Entity public class Parent
{
    private Collection<Child> children;

    @OneToMany(mappedBy = "parent")
    public Collection<Child> getChildren()
    { return children; }
    ...
}

@Entity public class Child
{
    private Parent parent;
    @ManyToOne
    public Parent getParent()
    { return this.parent; }
    ...
}
```

---

---

# ***Bidirectional Relationships***

- No CMR
- Manually reassign a child's parent in a bidirectional one to many



```
@Entity public class Order implements Serializable
{
    private Collection<LineItem> lineItems;

    @OneToMany(mappedBy = "order")
    public Collection<LineItem> getLineItems()
    { return lineItems; }

    private void setLineItems(Collection<LineItem> lineItems)
    { this.lineItems = lineItems; }

    public void addLineItem(LineItem lineItem)
    {
        if (lineItem.getOrder() != null)
            lineItem.getOrder().getLineItems().remove(lineItem);

        this.lineItems.add(lineItem);
        lineItem.setOrder(this);
    }
    ...
}
```

---

---

# *Relationships: Many to Many*

- Specify a collection (using generics) on each entity
  - Annotate each getter with `@ManyToMany`
    - The owning side must also define the `mappedBy` attribute
  - EJB3 will create the associative table for you
- 
-

```
@Entity public class Student implements Serializable
{
    private Collection<Teacher> teachers;

    @ManyToMany
    public Collection<Teacher> getTeachers()
    { return teachers; }
    ...
}
```

```
@Entity public class Teacher implements Serializable
{
    private Collection<Student> students;

    @ManyToMany(mappedBy = "teachers")
    public Collection<Student> getStudents()
    { return students; }
    ...
}
```

---

---

# *Relationships: Fetching*

- FetchType.EAGER
  - Always fetch the child when the parent is loaded
- FetchType.LAZY
  - Only load the child on demand



# *Relationships: Cascading*

- CascadeType.PERSIST
    - Create child objects when the parent is created
  - CascadeType.REMOVE
    - Remove child objects when the parent is removed
  - CascadeType.MERGE
    - Update child objects when the parent is updated
  - CascadeType.ALL
    - Everything above
- 
-

# *Relationships: Collections*

- EJB3 defines the following collection types:
    - Collection
    - Set
    - Map
    - List & Array (JBoss extension)
      - Indicate order by @IndexColumn
  - Helpful Collection Annotations
    - @OrderBy, @Where, @BatchSize, @Check, @Cache, @Sort, @onDelete
- 
-

# *Embedded Objects*

- Allows you to map components to the owning entity's table
- No persistent identity
- Possible usages
  - Composite Primary Keys
  - MonetaryAmount
  - Address



```
@Entity public class LineItem implements Serializable
{
    private MonetaryAmount price;

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="amount"),
        @AttributeOverride(name="currency")})
    public MonetaryAmount getPrice()
    { return price; }
    ...
}
```

```
@Embeddable public class MonetaryAmount implements
    Serializable
{
    private BigDecimal amount;
    private Currency currency;
    ...
}
```

---

---

# Composite Primary Key

```
@EmbeddedId({
    @AttributeOverride(name="ssn"),
    @AttributeOverride(name="tshirtsizesize")})
public PersonPK getPk()
{ return pk; }

@Embeddable
public class PersonPK implements Serializable
{
    private String ssn;
    private String tshirtsizesize;
    ...
}
```

---

---

# Secondary Tables

- Entities can span multiple database tables
- Simple: same primary key names
- Complex: requiring joins

```
@Entity
@SecondaryTables({
    @SecondaryTable(name="EMP_DETAIL"),
    @SecondaryTable(name="EMP_HIST")})
public class Person {...}
```



# *Entity Inheritance*

- Three Strategies
  - Single table Strategy
  - Table per class Strategy
  - Join Strategy



# *Single Table Strategy*

- Place all the attributes of the entire heirarchy in a single table
  - Pros
    - Easy to query
    - High performance, since no joins are required
  - Cons
    - Several null columns
    - Cannot use NOT NULL database constraints where children differ on nullability
- 
-

# *Table Per Class Strategy*

- Store each non abstract class in its own database table
  - Pros
    - Can enforce nullability
    - No excess columns
  - Cons
    - Complicated SQL requiring joins for simple operations
    - Problematic polymorphism
      - Requires multiple SQL selects
- 
-

# *Join Strategy*

- Store each vertical layer of the hierarchy in a database table
  - Pros
    - Optimal storage efficiency
    - Completely normalized data model
  - Cons
    - Complex joins for simple queries
    - Difficult ad hoc reporting
    - Unacceptable performance
- 
-

# ***EJB/QL***

- Virtually identical to HQL (Hybernate Query Language)
  - Object oriented query language
  - Query constructs reference Object getters instead of database columns
  - Polymorphic queries
  - Allows for early fetching, regardless of fetch strategy defined
- 
-

# EJB/QL

```
from Item
```

```
from Item i where i.name like '%ejb%'
```

*(not a distinct collection)*

```
from Item i join fetch i.bids
```

*(no join required!)*

```
from Item i join fetch i.bids b where b.id = 15
```

*(join is required)*

```
from Item i join fetch i.bids b where b.amount=15
```

*(implicit joins are discouraged)*

```
from Item i where i.bidder.address.city="Boulder"
```



# EJB/QL continued

*(Theta joins are supported)*

```
from Item i, Bid b
```

```
where i.id = b.item.id and b.name like "%myitem%"
```

*(Projection)*

```
select item from Item item
```

```
join item.bids bid
```

```
where bid.amount > 100
```

*(subqueries depend on underlying RDBMS)*

```
from Bid bid where bid.amount+1 >=
```

```
    (select max(b.amount) from Bid b;
```

*(returns an Object[])*

```
select item.id, item.name, bid.amount
```

```
from Item item join item.bids bid
```

```
where bid.amount > 100
```

---

---

# *EJB/QL: Dynamic Instantiation*

- Used to create report queries
- Create a POJO to store the values

```
select new RowItem(foo.id, foo.name, bar.name)
from Foo foo join foo.bars bar
```



# EJB/QL

- Aggregation
    - Count()
    - Min()
    - Max()
    - Sum()
    - Avg()
  - Group By
  - Having
  - Order By
- 
-

# *Criteria API (JBoss extension)*

- Cast the EntityManager to the Hibernate Session and use the Hibernate Criteria API
  - Very efficient dynamic queries
  - Object Oriented styled query writing
  - Somewhat more limited than EJB/QL but appropriate for certain situations such as search forms



# *Advanced Entity Techniques*

- Native SQL Queries
  - Stored Procedures
  - Query caching using the JBoss distributed data cache
  - Extended Persistence Contexts
    - Store an extended EntityManager in a Stateful session bean
    - @FlushMode(FlushModeType.NEVER)
    - Flush the transaction in the @Remove
- 
-

```
@Stateful
@Remote(ShoppingCart.class)
public class ShoppingCartBean implements ShoppingCart
{
    @PersistenceContext(type=PersistenceContextType.EXTENDED)
    EntityManager em;
    @EJB StatelessLocal stateless;
    private Customer customer;

    public long createCustomer() {
        customer = new Customer();
        customer.setName("William");
        em.persist(customer);
        return customer.getId();
    }

    @FlushMode(FlushModeType.NEVER)
    public void never()
    { customer.setName("Bob"); }

    @Remove public void checkout()
    { em.flush(); }
}
```

---

---

# *Moving On...*

- Dependency Injection & JNDI
- Callbacks
- Interceptors
- Security
- Deployment



# *Dependency Injection*

- Used to replace ejb-refs and resource-refs
  - Inject on fields or setter methods
  - Can inject anything registered in JNDI
  - Use @EJB to inject EJBs
  - Use @Resource to inject datasources
    - Also used for:
      - @Resource javax.ejb.SessionContext ctx;
      - @Resource javax.ejb.TimerService timer;
      - @Resource javax.ejb.UserTransaction ut;
  - @Home for EJB2.x home interfaces
- 
-

# Injection Samples

- Inject Foo into a field

```
@EJB private Foo foo;
```

- Inject Bar into a setter

```
private Bar bar;
```

```
@EJB(beanName="com.mycompany.BarBean")
```

```
public void setBar(Bar bar)
```

```
{ this.bar = bar; }
```

- Inject a Resource

```
@Resource(name="MyDataSource")
```

```
private javax.sql.DataSource myDataSource;
```



# JNDI

- All Beans are available via JNDI
    - Local
    - Remote
  - JBoss uses the interface name by default
    - `com.foo.bar.MyBean.class.getName()`
  - All injected dependencies create an entry in the JNDI ENC (`java:comp/env`)
    - JBoss actually uses `java:comp.ejb3/env`
- 
-

# Callback Intro

- Callbacks are not compulsory
- To create callbacks, either:
  - Annotate any method with the appropriate callback annotation
    - Noarg method can be named anything
  - Extract callbacks into a separate class.  
Annotate the bean with the `@EntityListener` annotation
    - Callback listener methods must take a single parameter which is the bean being called back



# Callback Example

```
@Stateless public class MyStatelessServiceBean
    implement MyStatelessService
{
    public String doSomething() {
        System.out.println("Do Something");
    }

    @PostConstruct
    public void postConstruct()
    {   System.out.println("PostConstruct"); }

    @PreDestroy
    public void preDestroy()
    {   System.out.println("PreDestroy"); }
}
```

---

---

# ***SLSB Callbacks***

- @PostConstruct
  - Called after dependency injection
- @PreDestroy
  - Immediately before removing from pool or bean destruction



# ***MDB Callbacks***

- @PostConstruct
  - Called after dependency injection
- @PreDestroy
  - Immediately before removing from pool or bean destruction



# *SFSB Callbacks*

- @PostConstruct
    - Called after dependency injection
  - @PreDestroy
    - Before removing from pool or bean destruction. This is called *before* the @Remove method
  - @PostActivate
  - @PrePassivate
- 
-

# *Entity Bean Callbacks*

- @PrePersist
  - @PostPersist
  - @PreRemove
  - @PostRemove
  - @PreUpdate
  - @PostUpdate
  - @PostLoad
- 
-

# *Interceptors*

- Surround method invocations on Session Beans and MDBs
  - Intercepts all method calls to the bean, passing it the `InvocationContext`
    - Allows you to determine which method was intercepted and the parameters
  - Interceptors may be defined within the class or in another class
  - Interceptor chains are allowed
- 
-

# Interceptors, part 2

@AroundInvoke

```
public Object myMethod(InvocationContext ctx)  
    throws Exception { return ctx.proceed(); }
```

- Ordering
  - Interceptors annotation on the Bean
  - @AroundInvoke inside the Bean



# Interceptor Example

```
@Stateless public class MyStatelessServiceBean
    implement MyStatelessService
{
    public String doSomething() {
        System.out.println("Do Something");
    }

    @AroundInvoke
    public Object logIt(InvocationContext ctx)
        throws Exception
    {
        System.out.println("Logging...");
        return ctx.proceed();
    }
}
```

---

---

# Security

- `@javax.annotation.security.RolesAllowed`
    - *Method level annotation*
  - `@javax.annotation.security.PermitAll`
    - *Method level annotation*
  - `@javax.ejb.RunAs`
    - *Class level annotation*
  - `@org.jboss.ejb3.security.SecurityDomain`
    - *Defines the JAAS realm to authenticate and authorize*
- 
-

# *Deployment*

- .par file
    - /META-INF/persistence.xml
      - Database dialect
      - JNDI lookup for the datasource
      - Additional attributes
  - All annotated EJBs will be automatically discovered and deployed
  - JBoss will also merge in any .hbm.xml files
- 
-

***EJB3***

Thank you

Darin Manica  
[darin@manica.org](mailto:darin@manica.org)

